International Journal of Computer Science Engineering and Information Technology Research (IJCSEITR)
ISSN(P): 2249-6831; ISSN(E): 2249-7943
Vol. 4, Issue 2, Apr 2014, 259-276
© TJPRC Pvt. Ltd.

DEVELOPMENT OF AN AUTOMATED FRAMEWORK TO RESOLVE SOFTWARE TESTING ISSUES

CHITTINENI ARUNA[1] & R. SIVA RAM PRASAD[2]

[1]Research Scholar, Acharya Nagarjuna University, Assistant Professor, Department of CSE, KKR & KSR Institute of Technology and Sciences, Guntur, Andhra Pradesh, India

[2]Research Guide, Department of CSE and Head, Department of IBS, Acharya Nagarjuna University, Guntur, Andhra Pradesh, India

ABSTRACT

The growing importance of Software in the present scenario can be attributed to the fact that software is utilized in several different ways for different issues to provide different types of solutions. The subject of Software Engineering is so vast and varied that it encompasses several areas such as manufacturing of Refrigerators, Air Conditioning Systems, Automobiles, Space Engineering, Wireless and Mobile Communications. Therefore, in most cases, software usually need to adhere to a specifications so that they can perform to produce only those results that are expected and enunciated.

In general, keeping in view of the present scenario, a software professional goes through a certain process to establish that the software conforms to a given specification. This process, verification and validation (V & V), ensures that the software conforms to its specification and that the customers ultimately receive what they ordered. Software testing is one of the techniques to use during V & V. To be able to use resources in a better way, computers should be able to help out in the "art of software testing" to a higher extent, than is currently the case today. The main issue is to not retrench human resources from the process of software testing itself altogether but to consider the fact that software engineering is an art and science of identifying suitable solutions to critical problems by not just involving computers alone to solve the testing issues but software engineers should themselves participate and come out with better and suitable solutions to those problems where computers are clearly not that good to provide appropriate solutions.

This research work presents a systematic and methodical approach aimed at examining, classifying and improving the concept of automated software testing and is built upon the assumption that software testing could be automated to a higher extent. Throughout this thesis an emphasis has been put on "real life" applications and the testing of these applications.

One of the contributions in this thesis is the research aimed at uncovering different issues with respect to automated software testing. The research is performed through a series of case studies and experiments which ultimately also leads to another contribution—a model for expressing, clarifying and classifying software testing and the automated aspects thereof. An additional contribution in this thesis is the development of framework which in turns acts as a broad substratum for a framework for object message pattern analysis of intermediate code representations. This is achieved keeping in view of the procedures relating to code optimization and effective code generation at all levels.

The results that are expected in this thesis indicate how software testing can be improved, extended and better classified with respect to automation aspects. The main contribution lays in the investigations and the improvements related issues with regard to automated software testing. A comprehensive study has been made by the researchers to tackle the very crust of the problem of software testing methodologies.

It has to be noted that testing software in an automated way has been a goal for researchers and industry during the last few decades. Nevertheless, the success rate has not been readily available. Some tools that are partly automated have evolved, while at the same time the methodologies for testing software has improved, thus leading to an overall improvement. Even today, much of the software testing procedures are performed manually which is not reliable, prone to errors, inefficient and also a costly affair.

Even today several questions still remain unanswered. For instance, we have the question of when and how to stop the process of testing software when is it not economically feasible to continue to test software? It is very much evident that investing so much of time and money for testing an application which will be used only for a few times, in a non-critical environment, is probably a waste of resources.[1] At the same time, when software engineers develop software that will be placed in a critical domain and extensively used, an answer to that question needs to be found.

Secondly, there is the problem of resources. It is to be noted that small- and medium-sized companies are today, as always, challenged by their resources, or to be more precise, the lack thereof. Deadlines must be kept at all costs even when, in some cases, the cost turns out to be the actual reliability of their products. In addition to the growing complexity of software, the size of the problem becomes more and more complex and huge.

Keeping in view of the question of complexity and to be more precise based on the fact that software has grown in complexity and size which have become very much part of the problem of software testing and the type of research work that needs to be done to improvise on these issues is to be given a methodical and systematic approach.

KEYWORDS: Structural Testing, Commercial-off-the-Shelf, Component Based Software Engineering, Automated Software Testing, Empirical Evaluation, Verification and Validation

INTRODUCTION

Ironically, software systems have been growing at an amazing speed with the growth and development of technology in the frontier areas of research which had greatly posed as a challenge to the software engineers as far as automation and semi-automation of software testing procedures are concerned. It is to be noted that software testing procedures are time consuming, need more and more of human intervention and are also costly.

Anyways, not everything in this particular area can be painted black. It may be appreciated that newer programming languages, tools and techniques that provide better possibilities to support testing have been released. In addition to that, development methodologies are being used, that provide a software engineering team ways and means of integrating software testing into their processes in a more natural and non-intrusive way.

As such, industry uses software testing techniques, tools, frameworks, etc. more and more today. However, unfortunately there are exceptions to this rule. There is a clear distinction between, on the one hand, large, and on the other hand, small- and medium-sized enterprises. Small- and medium-sized enterprises seem to experience a lack of resources to a higher degree than large enterprises and thus reduce and in some cases remove the concept of software testing all together from their software development process. Hence the need and requirement for introducing and improving automated software testing is even clearer in this case.

The aim of the research presented in this research paper is to improve software testing by increasing its effectiveness and efficiency. Effectiveness is improved by combining and enhancing testing techniques, while the factor of efficiency is increased mainly by examining how software testing can be automated to a higher extent. By improving effectiveness and efficiency in software testing, time can be saved and thus provide small software development companies the ability to test their software to a higher degree than what is the case today. In addition, to be able to look at automated aspects of software testing, definitions need to be established as is evident in this dissertation. Consequently, in this research work, a model with a number of definitions is presented which in the end helps in creating a well-defined framework which should be constructed to support a higher degree of automation.

The main contribution of this research work is in improving and classifying software testing and the automated aspects thereof. Several techniques are investigated, combined and in some cases improved for the purpose of reaching a higher effectiveness. While looking at the automated aspects of software testing a model is developed wherein a software engineer, software tester or researcher can classify a certain tool, technique or concept according to their level of automation. The classification of several tools, techniques and concepts, as presented in this dissertation, implicitly provide requirements for a future automated framework with a high degree of automation-a framework which in addition is presented in this dissertation as well.

The research as put forward in this thesis is focused on analyzing, improving and classifying software testing, especially the automated aspects thereof. Software testing is, in our opinion:

"the process of ensuring that a certain piece of software item fulfills its requirements".

Automation aspects in software testing, on the other hand, is focussed on keeping human intervention to a minimum as far as possible. In order to test a software item's requirements, a software engineer first needs to understand the basics of software quality and for this a fairly good knowledge of the underlying basic concepts is necessary to provide better results.

SOFTWARE QUALITY

What are the quality aspects a software engineer must adhere to, with respect to software development? It is not an easy question to answer since it varies, depending on what will be tested, i.e. each software is more or less unique, although most software have some common characteristics.

Some quality aspects, which can be found by examining today's literature are:

- **Reliability**: The extent with which a software can perform its functions in a satisfactory manner, e.g. an ATM which gives Rs.200 bills instead of Rs.100 bills is not reliable, neither is a space rocket which explodes in mid-air.
- **Usability:** The extent with which a software is practical and appropriate to use, e.g. a word processor where the user needs a thick manual to be able to write a simple note, possesses bad usability.

- **Security:** The extent with which a software can withstand malicious interference and protect itself against unauthorized access, e.g. a monolithic design can be bad for security. [2]
- **Maintainability:** The extent with which a software can be updated and thus ad-here to new requirements, e.g. a software that is not documented appropriately is hard to maintain.
- **Testability:** The extent with which a software can be evaluated, e.g. a bloated or complex design leads to bad testability.

Of course there exist more characteristics, e.g. portability, efficiency or completeness, but nevertheless, if a software engineer in a small- or medium-sized project would, at least, take some of the above aspects into account when testing software, many faults would be uncovered. It might be worth mentioning that going through all these attributes, time and again, is a time-consuming task, but when an engineer has gone through this process several times he will eventually gain a certain amount of knowledge and thus be able to fairly quickly see which attributes are essential for a particular software item.[3]

However, needing to re-invent the wheel, is something that should be avoided. Fortunately, The International Organization for Standardization has collected several of these attributes, i.e. quality aspects that a software engineer could test for, in the ISO 9126 standard. Nonetheless, all of these attributes affect each individual software in a unique way, thus still putting demands on a software engineer to have intricate knowledge of many, if not all, characteristics.

SOFTWARE TESTING

Software testing, which is considered to be a part of the verification and validation (V & V) area, has an essential role to play when it comes to ensuring a software’s implementation validity to a given specification. One common way to distinguish between verification and validation is to ask two simple questions [267]:

- Verification-are we building the product right?
- Validation-are we building the right product?

First of all, software testing can be used to answer the question of verification, e.g. by ensuring, to a certain degree, that the software is built and tested according to a certain software testing methodology, we can be assured that it has been built correctly. Secondly, by allowing customers and end-users to test the software currently being.

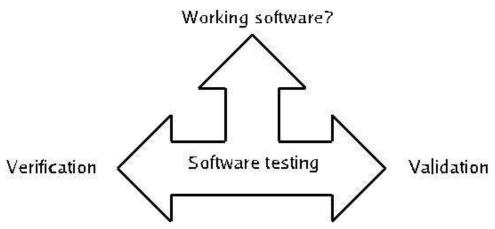

Figure 1: Software Testing and V & V

Hence, the way to view software testing would be to picture it being a foundation stone on which V & V is placed upon, while at the same time binding V & V together as shown in figure above.

Unfortunately, software testing still, by large, inherit a property once formulated by Dijkstraas:

"Program testing can be used to show the presence of bugs, but never to show their absence!"

The quote, is often taken quite literarily to be true in all circumstances by software engineering researchers but, obviously, depending on the statistical significance one would want to use, the above quote might very well not be true.

To sum it up, software testing is the process wherein a software engineer can identify e.g. completeness, correctness and quality of a certain piece of software. In addition, software testing is traditionally divided into two areas: white box and black box testing.

WHITE BOX

White box testing, structural testing or glass-box testing as some might prefer calling it, is actually not only a test methodology, but also a name that can be used when describing several testing techniques, i.e. test design methods. The lowest common denominator for these techniques is how an engineer views a certain piece of software.[4]

In white box testing an engineer examines the software, using knowledge concerning the internal structure of the software. Hence, test data is collected and test cases.

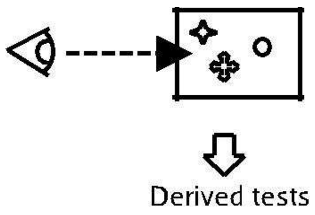

Figure 2: The Transparent View Used in White Box Testing gives a Software Engineer Knowledge about the Internal Parts Which are Written Using this Knowledge

Today white box testing has evolved into several sub-categories. All have one thing in common, i.e. how they view the software item. Some of the more widely known white box strategies are coverage testing techniques:

- Branch coverage.
- Path coverage.
- Statement coverage.

Branch Coverage is a technique to test each unique branch in a given piece of code. For example, each possible branch at a decision point, e.g. switch/case statement, is executed at least once, thus making sure that all reachable code is executed in that limited context.

Path Coverage, on the other hand, deals with complete paths in a given piece of code. In other words, every line of source code should be visited at least once during testing. [5] Unfortunately, as might be suspected, this is very hard to achieve on software that contains many lines of code, thus leading to engineers using this technique mostly in small and well delimited sub-domains e.g. that might be critical to the software's ability to function.

Statement Coverage's aim is to execute each statement in the software at least once. This technique has reached favorable results, hence the previous empirical validation makes this technique fairly popular. It is on the other hand questionable if this particular technique can scale reasonably, thus allowing a software tester to test larger software items.

Even though white box testing is considered to be well-proven and empirically validated, it still has some valid drawbacks in that it is not the silver bullet when it comes to testing. Experiments have shown and later that static code reading, which is considered to be a rather costly way to test software, is still cheaper than for example statement testing, thus making certain researchers question the viability of different coverage testing techniques . Worth pointing out in this case is that a coverage technique does not usually take into account the specification-something an engineer can do during formal inspection.

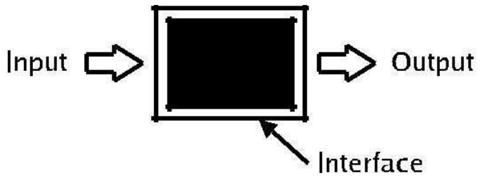

Figure 3: In Black Box Testing, the External View is What Meets the Eye

BLACK BOX

Black box testing, behavioral testing or functional testing, is another way to look at software testing. In black box testing the software engineer views, not the internals but instead the externals, of a given piece of software[6] The interface to the black box and what the box returns and how it correlates to what the software engineer expects it to return, is the essential corner stone in this methodology.

In the black box family several testing techniques do exist. They all disregard the internal parts of the software and focus on how to pass different values into a black box and check the output accordingly. The black box can, for example, consist of a method, an object or a component. In the case of components, both Commercial-Off-The-Shelf (COTS) and 'standard' components, can be tested with this approach (further clarifications regarding the concept of components are introduced.

Since many components and applications are delivered in binary form today, a software engineer does not usually have any choice but to use a black box technique. Looking into the box is simply not a viable option, thus making black box testing techniques useful in e.g. component-based development (CBD). In addition to that, CBD's approach regarding the usage of e.g. interfaces as the only way to give access to a component, makes it particularly interesting to combine with a black box technique. Worth mentioning here is the concept of intermediate representations of code (e.g. byte code or the common intermediate language). By compiling code into an inter-mediate form, certain software testing techniques can be easily applied which formerly were very hard if not impossible to execute on the software's binary manifestation. [7]

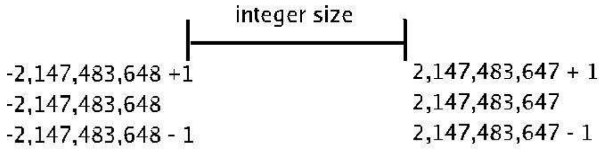

Figure 4: Using Boundary Value Analysis to Test an Integer

It needs to be mentioned that there are several techniques within the black box family. A very good collection of old and still valid black box techniques are available for a better understanding of the software testing processes. What follows next is a list of the most common techniques and a short explanation of each process as given:

- Boundary value analysis.
- Cause-effect graphing.
- Random testing.
- Partition testing (several sub-techniques exist in this field).

Boundary value analysis (BVA) is built upon the assumption that many, if not most, faults can be found around boundary conditions, i.e. boundary values. BVA has showed good results and is today considered to be a straightforward and relatively cheap way to find faults. As this thesis shows, BVA is among the most commonly utilized black box techniques in industry.[8]

In Figure 4, an example is shown on how a simple integer could be tested (note however that these numbers only apply to today's PCs). In other words, by testing the various boundary values a test engineer can uncover many faults that could lead to overflows and usually, in addition to that, incorrect exception handling.

Cause-effect graphing theory attempts to solve the problem of multiple faults in software.

[redacted]

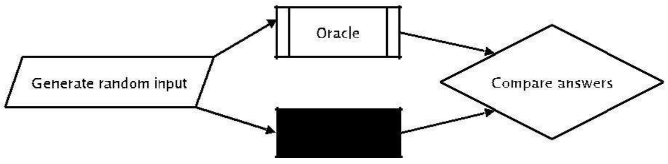

Figure 5: Random Testing Using an Oracle

This theory is built upon the belief that combinations of in-puts can cause failures in software, thus the multiple fault assumption uses the Cartesian product to cover all combination of inputs, often leading to a high yield set of test cases.

A software test engineer usually performs a variant of the following steps when creating test cases according to the cause-effect graphing technique:

- Divide specification into small pieces (also known as the divide and conquer principle).
- List all causes (input classes) and all effects (output classes).
- Link causes to effects using a Boolean graph.
- Describe combinations of causes/effects that are impossible.
- Convert the graph to a table.
- Create test cases from the columns in the table.

Random testing is a technique that tests software using random input. By using random input (often generating massive number of inputs) and comparing the output with a known correct answer, the software is checked against its specification. This test technique can be used when e.g. the complexity of the software makes it impossible to test every possible combination. Another advantage is that a non-human oracle [9], if available, makes the whole test procedure-creating the test cases, executing them and checking the correct answer-fairly easy to automate. But, as is indicated by this thesis, having a constructed oracle ready to use is seldom an option in most software engineering problems. In one way or another, an oracle needs to be constructed manually or semi-automatically.[10]

Partition testing equivalence class testing or equivalence partitioning, is a technique that tries to accomplish mainly two things. To begin with, a test engineer might want to have a sense of complete testing. This is something that partition testing can provide, if one takes the word ‘sense’ into consideration, by testing part of the input space.

Second, partition testing strives at avoiding redundancy. By simply testing a method with one input instead of millions of inputs a lot of redundancy can be avoided. [11]

The idea behind partition testing is that by dividing the input space into partitions, and then test one value in that partition, it would lead to the same result as testing all values in the partition. In addition to that, test engineers usually make a distinction between single or multiple fault assumptions, i.e. that a combination of inputs can uncover a fault.

But can one really be assured that the partitioning was performed in the right way? After have covered white and black box techniques, which software testing traditionally has been divided into, one question still lingers. What about the gray areas and other techniques, tools and frameworks that does not nicely fit into the strict division of white and black boxes?

COMBINATIONS, FORMALITY AND INVARIANTS

Strictly categorizing different areas, issues or subjects always leaves room for entities not being covered, in part or in whole, by such a categorization.[12] As such, this thesis will cover research which is somewhat outside the scope of how software testing is divided into black and white boxes. After all, the traditional view was introduced in the 60’s and further enforced in the late 70’s by Myers, so one would expect things to change.[13]

In this research work, combinations of white box and black box techniques will be covered wherein combinations of different testing techniques within one research area. Furthermore, formal methods for software testing will be introduced and an introduction to how test data generation research has evolved lately, will be covered.

First the concept of horizontal and vertical combinations will be introduced. A horizontal combination, in the context of this thesis, is defined as being a combination wherein several techniques act on the same level of scale and granularity. For example, combining several black and/or white box techniques for unit testing, is by us seen as a horizontal combination.[14] On the other hand, combining several techniques on different scale or granularity, e.g. combining a unit testing technique with a system testing technique is by us considered to be a vertical combination. Now why is it important to make this distinction? First, in our opinion, combinations will become more common (and has already started to show more in research papers the last decade). While the horizontal approach is partly covered in this thesis and in addition the vertical approach is not. Different vertical approaches have the last years starting to show up and more research will most likely take place in the future.[15]

Formal methods is not a subject directly covered by this thesis (but formal methods in software testing is on the other hand not easily fit into a black or white box and hence tended for here). Using formal methods an engineer start by, not writing code, but instead coupling logical symbols which represents the systems they want to develop.

The collection of symbols can then, with the help of e.g. set theory and predicate logic, be checked to verify that they form logically correct statements." In our opinion, formal methods is the most untainted form of Test Driven Development(TDD)and can lead to good results (Praxis High Integrity Systems claim one error in 10, 000 lines of delivered code. On the other hand, there are problems that need to be solved in the case of formal methods.]16] First, does formal methods really scale? Ross mentions a system containing 200, 000 lines of code, which is not considered to be sufficient for many types of projects. Second, to what extent are formal methods automated? In our opinion, more or less, not at all. The generation of the actual software item is automatic, but the generation needs specifications which are considered to be very cumbersome to write.

Finally, with respect to test data generation, a few new contributions have lately been published which has affected this thesis and most likely can have an impact on software testing of object-oriented systems by large in the future , in addition it is hard to categorize these contributions following a traditional view. Ernst et al. and Lam et al. has lately focused on generating likely invariants in object-oriented systems. A likely invariant is, to quote Ernst et al.

. . . a program analysis that generalizes over observed values to hypothesize program properties.

In other words, by collecting runtime data, an analysis can be performed where the properties of these values can be calculated to a certain degree (compare this to Claessen's et al. work on Quick Check where they formally set properties before-hand).[17] By executing a software item, an engineer will be able to, to put it bluntly, generate -values of an existing system being developed. As an example, suppose a software executes a method twice; with the integer input value 1 the first time, and 5 the second time. in this case (when accounting for the boundaries) are the values 1, 2, 3, 4, 5, i.e. we have input values (or likely invariants) for the values 1 and 5. Even though this is a very simple example it might give the reader an idea of the concept (using the definition of, as is done in the example, is not always this straightforward in real life).

The contributions regarding likely invariants by Ernst et al. and Lam et al. is decidedly interesting for two reasons.[18] First, they can generate likely invariants for complex types. Second, by having likely invariants a fully automated approach can be reached (an engineer does not need to formally define properties in advance). In other words, a fully automated approach for testing object-oriented, imperative, software systems might be realizable.

AIMS AND LEVELS OF TESTING

Our research work includes several methods and techniques for testing software which forms the basis of our research results. In this research work, an overview of, and some examples on, the different views of testing are given, thus providing an introduction to and understanding of the various methodologies that testing can adhere to. It has to be noted that our research work does not cover the concepts of static and dynamic analysis. This is rightly so since in this thesis these concepts are not viewed as testing techniques themselves but rather as supporting techniques that can be used for testing software. These two techniques approach software analysis in two different ways. In static analysis an engineer does not actually run the program while in dynamic analysis, data regarding the software behavior is collected during run-time.[19] Some of the techniques that can be used in dynamic analysis are profilers, assertion checking and run-time instrumentation, while static analysis uses tools, such as source code analyzers, for collecting different types of metrics. This data can then be used for example in detecting memory leaks or invalid pointers.

Apart from the above two concepts (dynamic and static) a system can, in addition, be seen as a hierarchy of parts, e.g. sub-systems/components, objects, functions and a particular line of code. Since a system or a piece of software can be rather large; testing small parts initially and then continuously climb up the pyramid, would make it possible to achieve a reasonably good test coverage on the software as a whole, without getting lost in the complexity that software can exhibit.

In an example of a five-stage testing process is given, which is illustrate in the figure below, It is important to keep in mind that this is only one example and usually a software testing process varies depending on several outside factors.[20] Nevertheless, software testing processes as used today, often follow a bottom-up approach, i.e. starting with the smallest parts and testing larger and larger parts, while top-down approach starts with testing larger parts first followed by testing smaller parts.

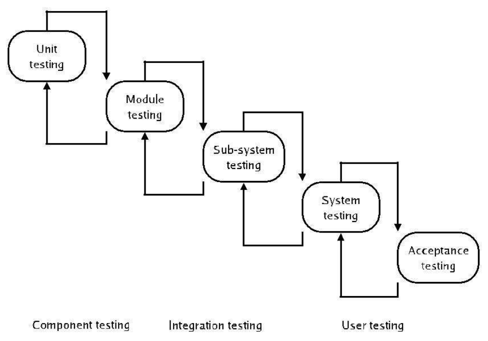

Figure 6: Bare-Bone Example of a Testing Process

Each and every stage in Figure 6 can be further expanded or completely replaced with other test stages, all depending on what the aim is in performing the tests. Nevertheless, the overall aim of all test procedures is of course to find run-time failures, faults in the code or, generally speaking, deviations from the specification at hand. However, there exists several test procedures that especially stress a particular feature, functionality or granularity of the software:

1. System functional testing	8.Installation Testing
2. Integration testing	9. Usability testing
3. Regression testing	10.Stability testing.
4. Load testing	11.Authorization testing
5. Performance testing	12.CustomerAcceptance Testing
6. Stress testing	13. Deployment Testing
7. Security testing	

In the end, it might be more suitable to use the word ‘aspect’ when describing the different test procedures. The phrase aspect-oriented testing illustrates in a better way keeping in view that a software test engineer is dealing with different aspects of the same software item.

TESTING IN OUR RESEARCH

This thesis focuses on the black box concept of software testing. Unfortunately the world is neither black nor white-a lot of gray areas do exist in research as they do in real life. To this end, the last part of this thesis takes into account the possibility to look 'inside' a software item. The software item, in this particular case, is represented in an intermediate language and thus suitable for reading and manipulating. Intermediate representations of source code is nowadays wide-spread (the notion of components is supported very much indeed by the the intermediate representations) and used by the Java Virtual Machine and the Common Language Runtime.

Indeed, the main reason for taking this approach is the view on software in general. We believe that Component-Based Software Engineering (CBSE) and Component-Based Development (CBD) will increase in usage in the years to come.

Since the word component can be used to describe many different entities a clarification might be appropriate for a view on how completely differently researchers view the concept of components.

[. . .] a self-describing, reusable program, packaged as a single binary unit, accessible through properties, methods and events.

In the context of this thesis, the word self-describing should implicitly mean that the test cases, which a component once has passed, need to accompany the component throughout its distribution life-time, preferably inside the component. This is seldom the case today. The word binary, on the other hand, indicates that a white box approach, even though not impossible, would be somewhat cumbersome to use on a shipped component-this is not the case as this thesis will show a technique is introduced wherein a white box approach is applied on already shipped components.

Finally, all software, as used in this thesis, is based on the imperative programming paradigm (whether object-oriented or structured) and can be considered as 'real life' software (and not small and delimited examples constructed beforehand to suite a particular purpose).

RESEARCH QUESTIONS

During the work on this thesis several research questions were formulated which the research then was based upon. The initial main research question that was posed for the complete research in this thesis was:

Main Research Question: How can software testing be performed efficiently and effectively especially in the context of small- and medium-sized enterprises?

In order to be able to address the main research question several other research questions needed to be answered first (RQ2–RQ10). In the following figure, the different re-search questions and how they relate to each other are depicted.

The first question that needed an answer, after the main research question was formulated, was:

RQ2: What is the current practice concerning software testing and reuse in small- and medium-sized projects?

By just putting the main research question might have been a question of no relevance. Thus, since this thesis is based upon the main research question, it was worthwhile taking the time to examine the current practice in different projects and see how soft-ware reuse and, especially, software testing was practiced. The answer to this research question will be found in our research work together with an analysis of how software testing is used in different types of projects.

To put it short, the answer to RQ2 divided the research, as presented in this thesis, into two areas covering effectiveness in software testing techniques and efficiency in software testing. To begin with, the research aimed at exploring the factor of effectiveness (RQ3–RQ6) while later focusing on efficiency and the automated aspects of software testing (RQ7–RQ10).

In order to examine if the current practice in software development projects was satisfactory for developing software with sufficient quality, RQ3 evolved into:

RQ3: Is the current practice, within software development projects, sufficient for testing software items?

The answer to RQ3 is to be found and provides us with meager reading with respect to the current practice in software projects. Additionally, the answer to RQ3 indicated that the answer to RQ2 was correct (regarding the poor status of software testing in many software development projects) and so, in addition, further shows the importance of the main research question.

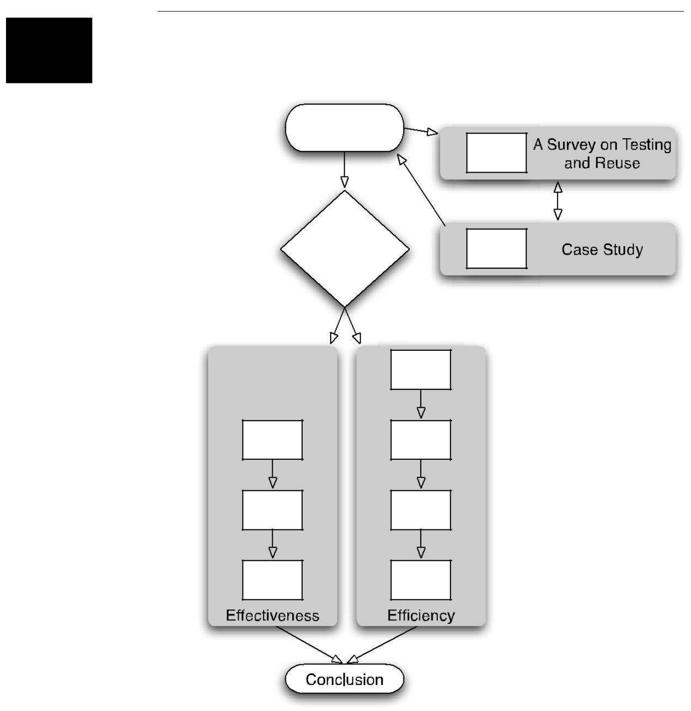

Figure 7: Relationship between Different Research Questions in this Thesis

Since a foundation for further research now had been established several research questions could be posed which in the end would help in answering the main research question.

RQ4: How can a traditional software testing technique (such as random testing) be improved for the sake of effectiveness?

The answer to RQ4 can be found in our research work which introduces new kinds of quality estimations for random testing and hence indirectly led to Research Question 5:

RQ5: How do different traditional software testing techniques compare with respect to effectiveness?

The answer to RQ5 can be found in our research work, which compares different traditional software testing techniques. The comparison in RQ5 eventually led to the question of combining different testing techniques:

RQ6: What is the potential in combining different software testing techniques with respect to effectiveness (and to some extent efficiency)?

At this stage in this thesis, the focus turns away from the factor of effectiveness and a full emphasis is put on the issue of efficiency. Since RQ2 indicated that there existed a shortage of resources for projects one of the conclusions was that software testing techniques not only need to be better at finding faults, but more importantly need to be automated to a higher degree and thus, in the long run, save time for the process' stake-holders.

Thus the following question was posed:

RQ7: What is the current situation with respect to automated software testing research and development?

The answer to RQ7 gave an approximate view of the status of automated software testing, but nevertheless was hard to formalize in detail due to the complexity of the research area and the share amount of contributions found. To this end, a model was developed which was able to formalize the area of software testing focusing, in the case of this thesis, especially on automated aspects.

RQ8: How can the area of software testing and its aspects be formalized further for the sake of theoretical classification and comparison?

The model with its accompanying definitions which partly is an answer to RQ8, was then used to classify, compare and elaborate on different techniques and tools:

RQ9: How should desiderata of a future framework be expressed to fulfill the aim of automated software testing, and to what degree do techniques, tools and frameworks fulfill desiderata at present?

Finally, the last part of this thesis focuses on the last research question:

RQ10: How can desiderata (as presented in RQ9) be implemented?

This thesis provides research results from implementing a framework for automated object message pattern extraction and analysis. Research Question 10, indirectly, provided an opportunity to: a) examine the possible existence of object message patterns in object-oriented software and, b) show how object message pattern analysis (from automatically instrumented applications) can be used for creating test cases.

Before any work on solving a particular research questions starts (a research question is basically a formalization of a particular problem that needs to be solved) a researcher needs to look at how the problem should be solved. To be able to do this, one must choose a research methodology.

RESEARCH METHODOLOGY

First of all, the research presented in this thesis aimed at using examples in the empirical evaluations which were used in industry, especially among small- and medium-sized projects. This, mainly because the research performed will hopefully, in the end, be used in this context. In addition to that, the academic community has endured some criticism for using e.g. simple 'toy' software, when trying to empirically validate theories.

Furthermore, the research as presented in this thesis always tried to have an empirical foundation, even when a theoretical model was developed.

To focus on the theory only and disregard an empirical evaluation would be, for our purposes, meaningless, especially so when the previous paragraph is taken into consideration. Empirical evaluations, of some sort, were always used as a way to investigate if a certain theory could meet empirical conditions, hence each step in this thesis was always evaluated.

Initially, in this thesis a qualitative approach was used with some additional quantitative elements. The results led us to the conclusion that more must be done by primarily, trying to improve current testing techniques and secondarily, looking at the possibilities at automating one or more of these techniques. To this end, an exploratory study was set up where a reusable component was tested in a straightforward way. The aim was to try to show that even basic testing techniques, e.g. unit testing, can uncover faults in software that had been reused and reviewed by developers. At the same time the study gave some indication on the validity of the various aspects of the survey.

The survey and the exploratory study provided indications that some areas could benefit from some additional research. Thus the following work was conducted:

- An improvement in how a software engineer might use random testing hence combining it with other statistical tools.
- A comparative study between two black box techniques, i.e. partition and random testing, giving an indication of the pros and cons of the respective techniques.

The research on improvements, with respect to random testing was performed using theoretical constructions which later were empirically evaluated, while the comparative study which was implemented by empirically evaluating a soft-ware item already used by industry. The methodology used in this research work is common in software engineering research.

Next, an empirical evaluation was performed where the following issues were researched:

- Strengths and weaknesses of different testing techniques.
- Combination of different testing techniques.

The focus was on researching the improved effectiveness of combining several testing techniques.

Arigorous methodology is applied for the purpose of outlining automated software testing research. It is appropriate to mention here that the literature study in no way claims to be exhaustive but instead attempts to take into account the significant areas of interest.

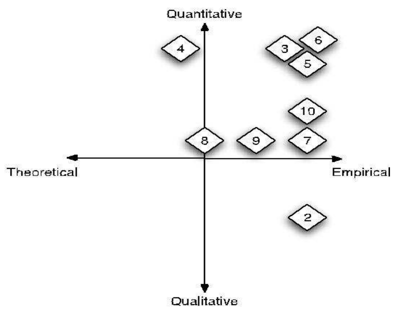

Figure 8: Methodologies Used in this Thesis

Our research work has a theoretical foundation which is then strengthened via an empirical evaluation. The model which is introduced is used to compare, classify and elaborate on different automated software techniques and then further, empirically, strengthened by the research survey in which certain aspects of the model is extensively used.

Finally, a case study is conducted where a framework is developed and used on different software items with the aim to extract software testing patterns automatically.

Figure 8 provides a rudimentary view of the methodologies as used in this thesis.

CONTRIBUTIONS OF THESIS

First, this thesis presents research on the current usage of different testing techniques in industry today, with a focus on small- and medium-sized enterprises. In addition to that some black box techniques are theoretically extended while at the same time being evaluated empirically. Special consideration is placed upon the future goal of automatically creating and executing different types of software testing techniques.

Second, a formal model with accompanying definitions, for classifying, comparing and elaborating on software testing and the automated aspects thereof is presented and validated in this thesis. The model is used for classifying several techniques and, in addition, the research work involves developing an automated software testing framework which is developed with the help of the model. The framework is implemented and validated against several ‘real life’ software items of considerable size.

Thus, the main contributions of this thesis are as follows:

- A view of the current state of practice in industry.
- A formal model for classifying software testing and the automated aspects thereof.
- A framework for supporting automated object message pattern analysis with the possibility to create test cases.
- Empirical evaluations of several black box techniques.

By looking at the testing techniques in use today, improvements could in the end be made, by:

- Improving different techniques’ effectiveness even more.

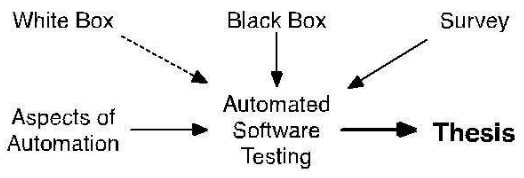

Figure 9: Diagrammatic Representation of Research Areas this Thesis Covers

The dashed line indicates a weak connection.

- Combining several techniques, thus reaching an even higher effectiveness and/or efficiency.
- Creating test cases and test suites automatically and/or semi-automatically by adapting the proposed framework.

- Using the proposed model to classify and compare software testing tools when needed, i.e. in industry to buy the right tool, or for researchers when comparing different tools for the sake of research.

FUTURE RESEARCH

The research in this thesis paves the way for several questions that need to be answered and in addition identify more research areas needed for further investigation. In this research work, the two most interesting areas which in our opinion has the highest potential to increase efficiency in projects are covered and given due importance keeping in view of the solutions that are provided to further improvise on the issues pertaining to software testing processes. It is worth mentioning, in this context, that there of course exist many other interesting research questions that need to be answered and for which our research will greatly help in providing a path to many software testers to develop and adopt suitable methods to overcome critical issues in the field of automated software testing.

CONCLUSIONS & RESULTS

There should be an immediate focus on the framework, a few more exist. For example, the following matters should be examined in the future:

- How common are the patterns, and how many patterns can we find in different types of applications?
- What pattern recognition strategies can be applied on the data collected from applications?
- Are there more ‘traditional’ testing techniques that can be applied on-top of the framework to further increase effectiveness?
- Use runtime data as collected, and stored in the object database, for calculating likely invariants off-line instead of doing it during runtime.
- How can the framework be adapted to other integrated development environments?

A case study encompassing even more applications should be performed to answer the first point, while, in order to answer the second bullet, a survey of statistical pattern recognition should be performed. The third bullet, should be clarified by further experimentation in, and development of frameworks such as the one laid forward in this thesis. Finally, experiments and case studies should provide an answer to the last two bullets.

REFERENCES

1. J.-R. Abrial. The B-Book: Assigning Programs to Meanings. Cambridge University Press, New York, NY, USA, 1996.
2. The ACM Digital Library. http://portal.acm.org/dl.cfm, May 2006.
3. W. R. Adrion, M. A. Branstad, and J. C. Cherniavsky. Validation, Verification, and Testing of Computer Software. ACM Computing Surveys, 14(2):159–192, 1982.
4. L. van Aertryck, M. Benveniste and D. Le Métayer. CASTING: A Formally Based Software Test Generation Method. In Proceedings of the 1st International Conference on Formal Engineering Methods, page 101. IEEE Computer Society, 1997.

5. S. Agerholm and P. G. Larsen. A Lightweight Approach to Formal Methods. In FM-Trends 98: Proceedings of the International Workshop on Current Trends in Applied Formal Method, pages 168–183, London, UK, 1999. Springer-Verlag.

6. Agitar Software, Enterprise Developer Testing for Java. http://www. agitar.com, May 2006.

7. A. Aksaharan. cUnit. http://cunit.sf.net/, May 2006.

8. P. Ammann and J. Offutt. Using Formal Methods to Mechanize Category-Partition Testing. Technical Report ISSE-TR-93-105, Dept. of Information & Software Systems Engineering, George Mason University, USA, September 1993.

9. D. M. Andrews and J. P. Benson. An Automated Program Testing Methodology and its Implementation. In Proceedings of the 5th International Conference on Software Engineering, pages 254–261. IEEE Computer Society, 1981.

10. J. H. Andrews. A Case Study of Coverage-Checked Random Data Structure Testing. In ASE 2004: Proceedings of the 19th IEEE International Conference on Automated Software Engineering, pages 316–319, Linz, Austria, September 2004. IEEE Computer Society.

11. N. W. A. Arends. A Systems Engineering Specification Formalism. PhD thesis, Eindhoven University of Technology, The Netherlands, 1996.

12. C. Artho, H. Barringer, A. Goldberg, K. Havelund, S. Khurshid, M. Lowry, C. Pasareanu, G. Rosu, K. Sen, W. Visser, and R. Washington. Combining Test Case Generation and Runtime Verification. Theoretical Computer Science, 2004. To be published.

13. Avritzer and E. J. Weyuker. The Automatic Generation of Load Test Suites and the Assessment of the Resulting Software. IEEE Transactions on Software Engineering, 21(9):705–716, 1995.

14. E. R. Babbie. Survey Research Methods. Wadsworth Pub. Co., Inc., 1990.

15. J. Bach. Test Automation Snake Oil. Windows Tech Journal, October 1996.

16. V. R. Basili and R. W. Selby. Comparing the Effectiveness of Software Test-ing Strategies. IEEE Transactions on Software Engineering, 13(12):1278–1296, 1987.

17. B. Baudry, F. Fleurey, J.-M. Jézéquel, and Y. Le Traon. Computational Intel-ligence for Testing. NET Components. In Proceedings of Microsoft Summer Research Workshop, Cambridge, UK, September 9–11 2002.

18. B. Baudry, F. Fleurey, J.-M. Jézéquel, and Y. Le Traon. Genes and Bacteria for Automatic Test Cases Optimization in the .NET Environment. In Proceedings of the 13th International Symposium on Software Reliability Engineering, pages 195–206. IEEE Computer Society, 2002.

19. K. Beck. Test Driven Development: By Example. Addison-Wesley Pub. Co., Inc., USA, 2002.

20. K. Beck and E. Gamma. Test Infected: Programmers Love Writing Tests. Java Report, pages 51–66, 1998.